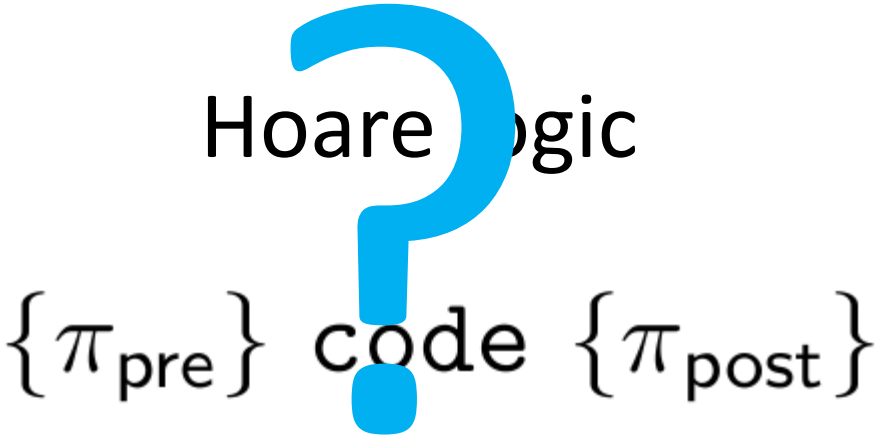
Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with Andrei Stefanescu and Chucky Ellison. Started with Wolfram Schulte at Microsoft Research in 2009

Question

… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?



Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings of L
- Program verifiers for L
  - Based on some other adhoc encodings of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings of L
- …

Example of C Program

- What should the following program evaluate to?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- According to C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**
  By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
- Informal manuals are not sufficient
  - Manuals typically have a formal syntax of the language (in an appendix)
  - Why not a formal semantics as well?

Motivation and Goal

- We are facing a semantic chaos
  - Operational, denotational, axiomatic, etc.
  - Problematic when dealing with large languages
- Why so many semantic styles?
  - Since none of them is ideal, they have limitations

- We want a powerful, unified foundation for programming language semantics and verification
  - One semantics to serve all the purposes!

Minimal Requirements for an Ideal Language Semantic Framework

- Should be expressive
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be executable
  - So we can test it and use it in tools (symb. exec.)
- Should be modular (and thus scale)
  - So each feature is defined once and for all
- Should serve as a program logic
  - So we can also prove programs correct with it

Current Semantic Approaches

- Structural Operational Semantics
  - Executable; not very modular; not appropriate for verification; only interleaving semantics
- Denotational Semantics
  - Reasonable trade-offs; not very executable (Norish's C semantics is not executable, and factorial(5) crushes Papaspyrou's C semantics); not very good for verification; bad for concurrency; expert knowledge
- Axiomatic Semantics (Floyd/Hoare logic)
  - Good for verification (sort of); not executable; not very expressive (extensions typically needed)
- Etc.

Towards a Better Semantic Approach

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- Expressive
  - Any logic can be represented in RL (it is reflective)
- Executable
  - Quite efficiently; Maude often outperforms SML
- Modular
  - Allows rules to only “match” what they need
- Can potentially serve as a program logic
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

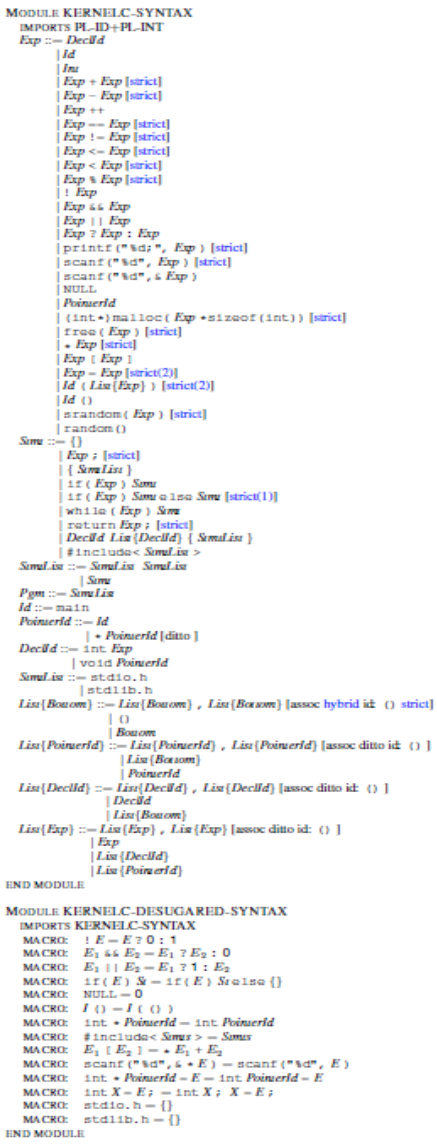
Rewriting Logic Semantics Project

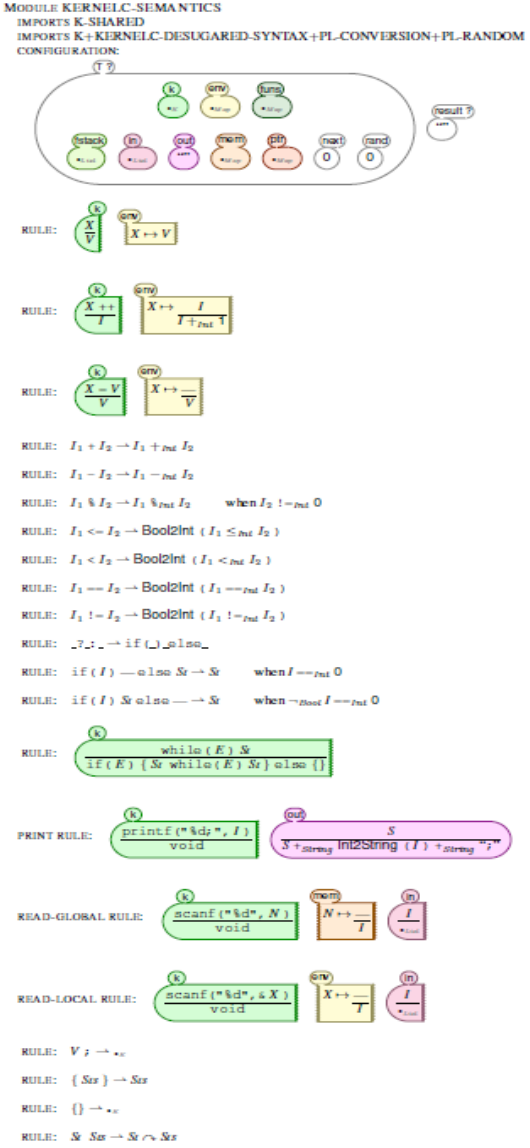
- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
  - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

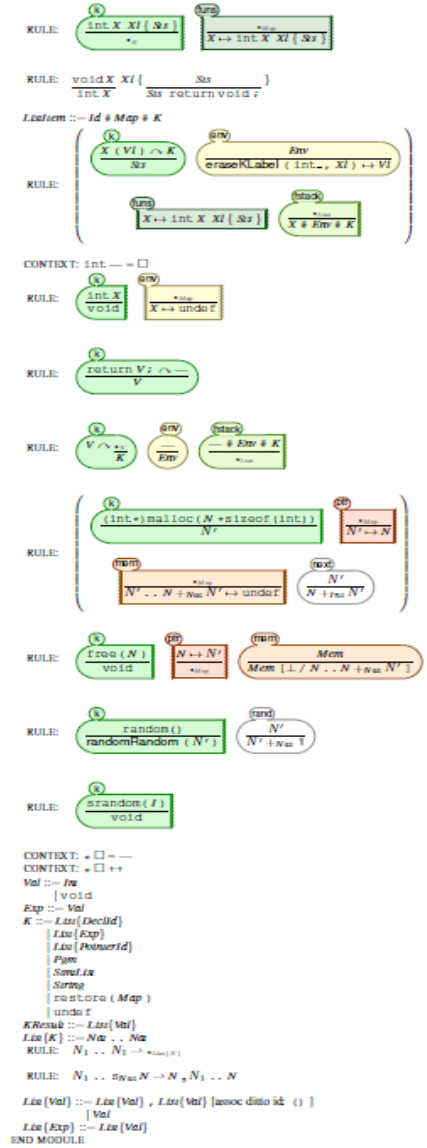
The K Framework

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching undergraduate courses
- Ideas:
  - Represent program configurations as a nested structure of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by rules (a small number, typically 1 or 2)

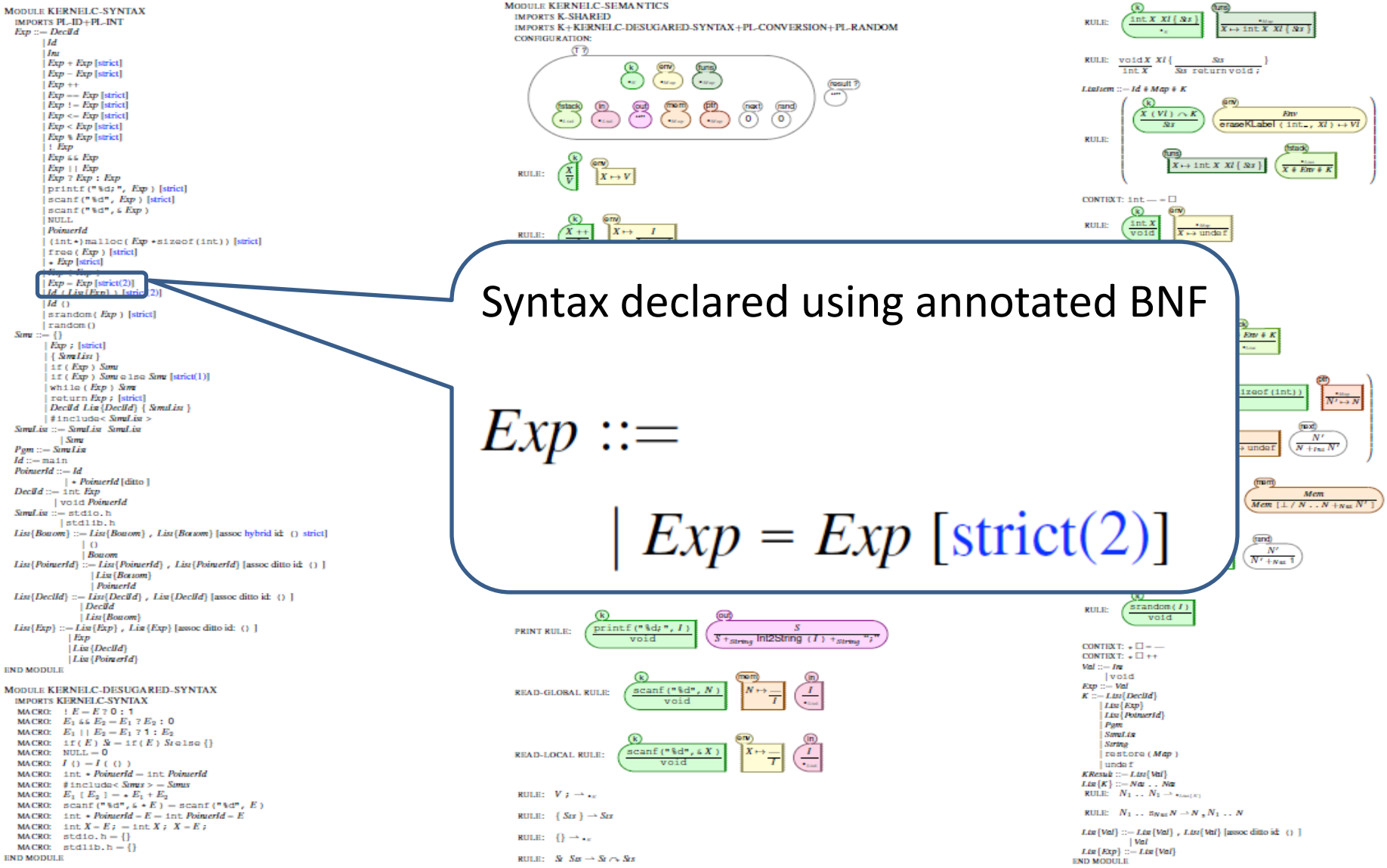
Complete K Definition of KernelC





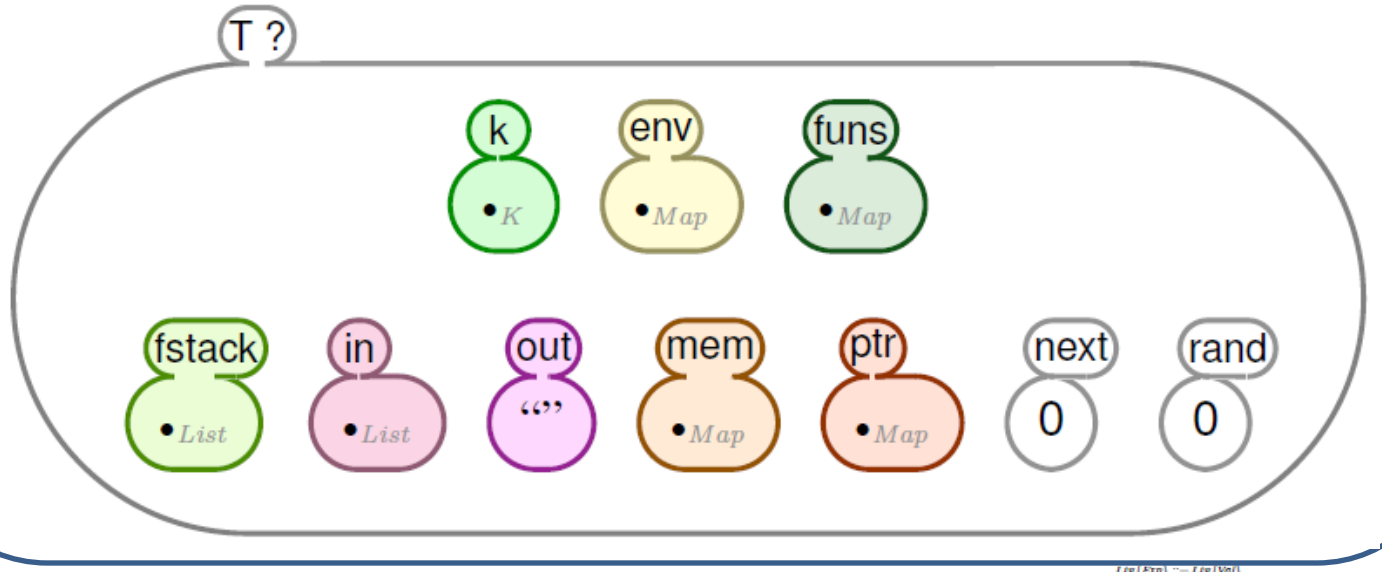


Complete K Definition of KernelC

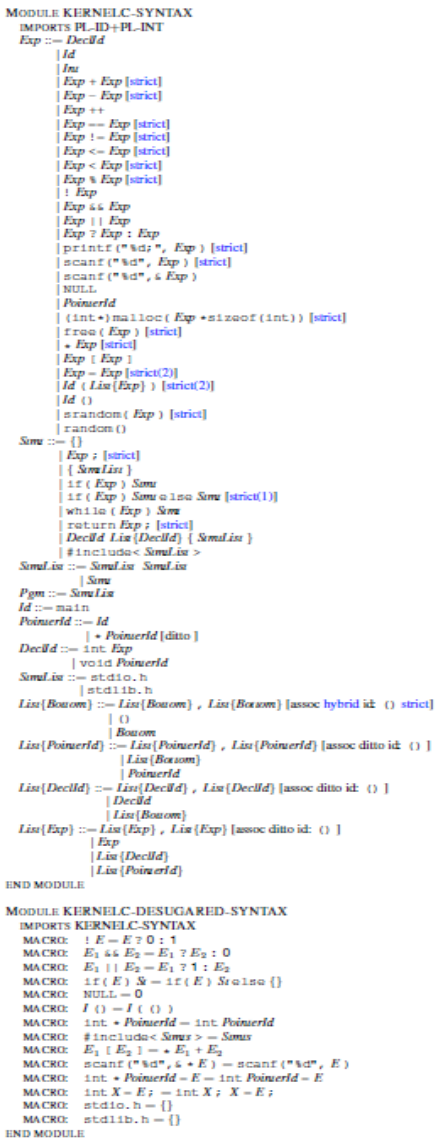


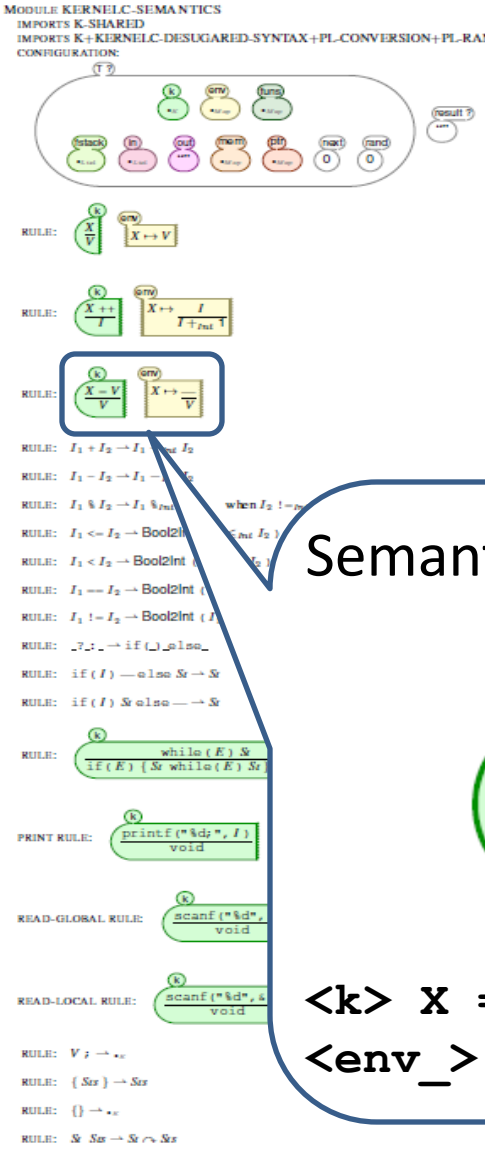
Complete K Definition of KernelC

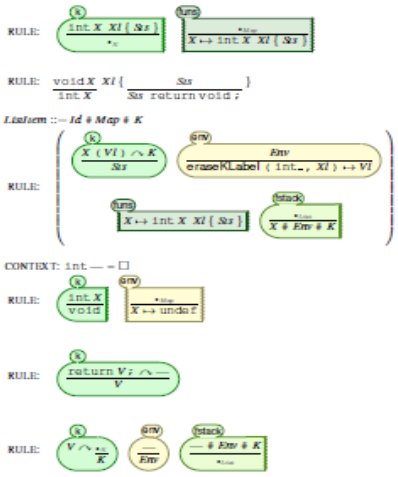
Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax



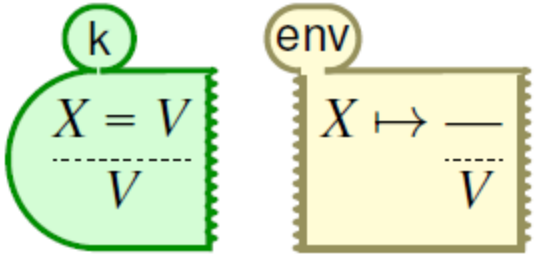
Complete K Definition of KernelC







Semantic rules given contextually




```
<k> X = V => V </_k>
<env_> X |-> (_ => V) </_env>
```

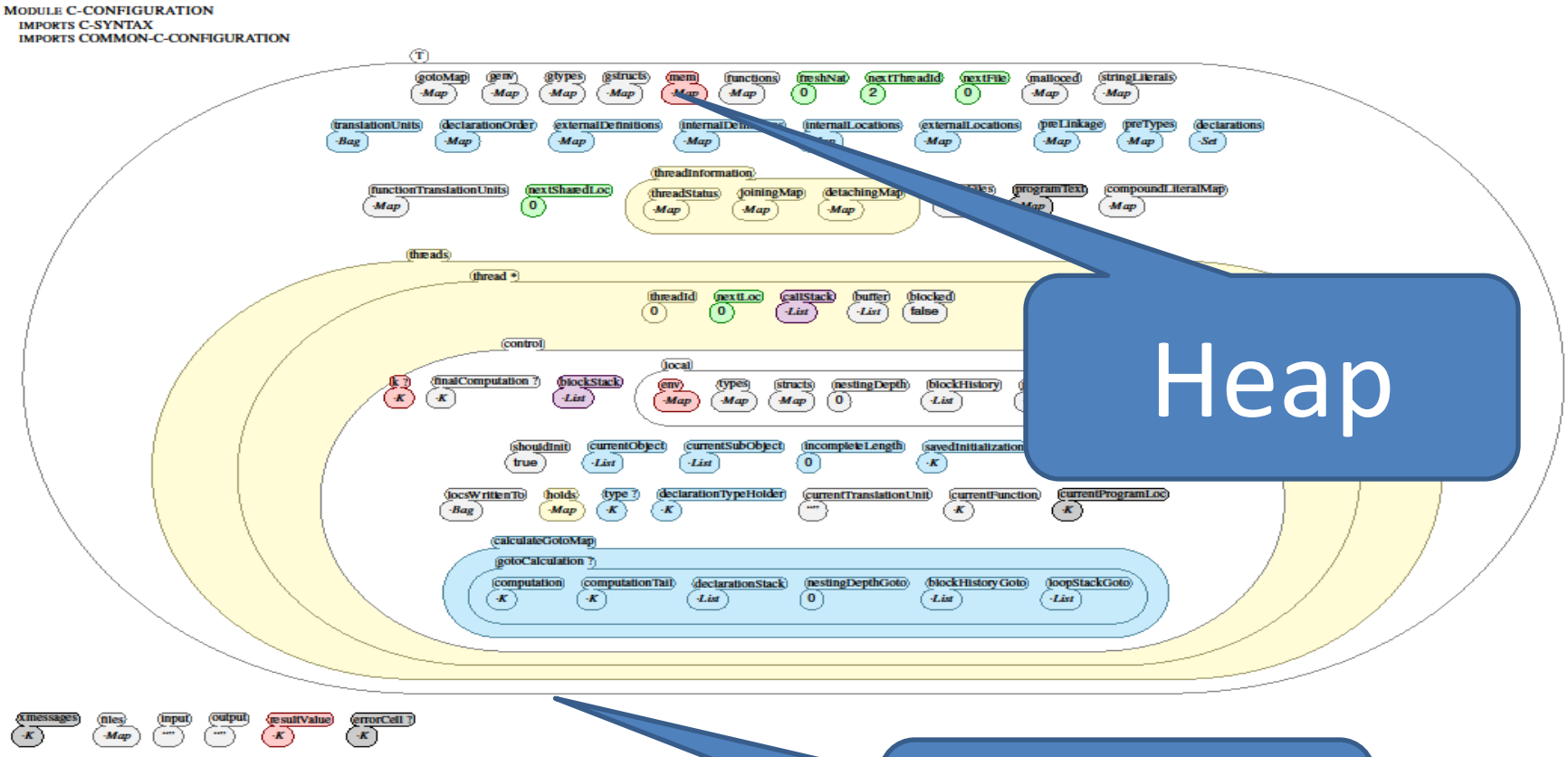
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C



Statistics for the C definition

- Syntactic constructs: 173
- Total number of rules: 812
- Total number of lines: 4688

- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test – passed 96% so far)
- The most complete formal C semantics
- Took more than 1 year to define …
  - Wouldn't it be uneconomical to redefine it in each tool?

Executable Semantics are Useful

- Compiler certification (Leroy's talk)
- Help language designers
- K semantics are currently compiled into
  - Maude, for execution, debugging, model checking
  - Latex, for human inspection and understanding
  - Soon to OCAML, for fast execution
- Can we use K semantics for program verification?
  - E.g., we want to use the C semantics unchanged for program verification; defining an alternative (axiomatic) semantics is very inconvenient and error prone!

Matching Logic = K + FOL

- A logic for reasoning about configurations
- Formulae
  - FOL over configurations, called patterns
  - Configurations are allowed to contain variables
- Models
  - Ground configurations
- Satisfaction
  - Matching for configurations, plus FOL for the rest

Examples of Patterns I

- x is bound to 5 and is the only variable in environment

$$\langle \mathrm{x} \mapsto 5 \rangle_{\mathsf{env}}$$

- x is bound to 5 in the current environment

$$\exists \rho \ (\langle \mathrm{x} \mapsto 5, \ \rho \rangle_{\mathsf{env}})$$

- x is bound to a non-negative value

$$\exists a \exists \rho \ (\langle \mathrm{x} \mapsto a, \ \rho \rangle_{\mathsf{env}} \ \wedge \ a \geq 0)$$

- x and y hold equal positive values

$$\exists a \exists \rho \ (\langle \mathrm{x} \mapsto a, \ \mathrm{y} \mapsto a, \ \rho \rangle_{\mathsf{env}} \ \wedge \ a > 0)$$

Examples of Patterns II

- x and y are aliased

$$\exists a \exists \rho \exists u \exists \sigma \; (\langle \mathrm{x} \mapsto a,\ \mathrm{y} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle a \mapsto u,\ \sigma \rangle_{\mathsf{heap}})$$

- x and y not aliased, and x points to larger value

$$\exists a \exists b \exists \rho \exists u \exists v \exists \sigma \; (\langle \mathrm{x} \mapsto a,\ \mathrm{y} \mapsto b,\ \rho \rangle_{\mathsf{env}}\ \langle a \mapsto u,\ b \mapsto v,\ \sigma \rangle_{\mathsf{heap}}\ \wedge u > v)$$

- x points to a list containing sequence A

$$\exists a \exists \rho \exists \sigma \; (\langle \mathrm{x} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle \mathsf{list}(a, A),\ \sigma \rangle_{\mathsf{heap}})$$

Examples of Patterns III

- x points to sequence A, and the reversed sequence A has been output

$$\exists a \exists \rho \exists \sigma \exists \omega \ (\langle \mathtt{x} \mapsto a,\ \rho \rangle_{\mathsf{env}}\ \langle \mathsf{list}(a, A),\ \sigma \rangle_{\mathsf{heap}}\ \langle \omega, \mathsf{rev}(A) \rangle_{\mathsf{out}})$$

- **`untrusted()`** can only be from **`trusted()`**

$$\exists s_1 \exists s_2 \ (\langle \mathtt{untrusted()} \rangle_{\mathsf{k}}\ \langle s_1, \mathtt{trusted()}, s_2 \rangle_{\mathsf{fstack}})$$

- Read/Write datarace (simplified)

$$\exists X \exists a \ (\langle X \cdots \rangle_{\mathsf{k}}\ \langle X = a \cdots \rangle_{\mathsf{k}})$$

Matching Logic vs. Separation Logic

- Matching logic achieves separation naturally, through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap; recall that the heap was only 1 out of the 75 cells in C's def.
- Matching logic stays within FOL, while separation logic extends FOL
  - Thus, we can use the existing SMT solvers, etc.

Matching Logic as a Program Logic

- Hoare style - not recommended

$$\{\pi_{\text{pre}}\}\ \texttt{code}\ \{\pi_{\text{post}}\}$$

  – One has to redefine the PL semantics – impractical

- Rewriting (or K) style – recommended

$$left[\texttt{code}] \rightarrow right$$

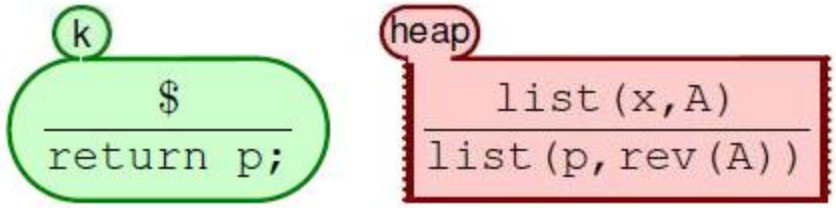
  – One can reuse existing K semantics – very good

Example – Reversing a list

```
struct listNode* reverse(struct listNode *x)
{
  struct listNode *p;
  struct listNode *y;
  p = 0 ;
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

$

- What is the K semantics of the reverse function?
- Let $ be its body



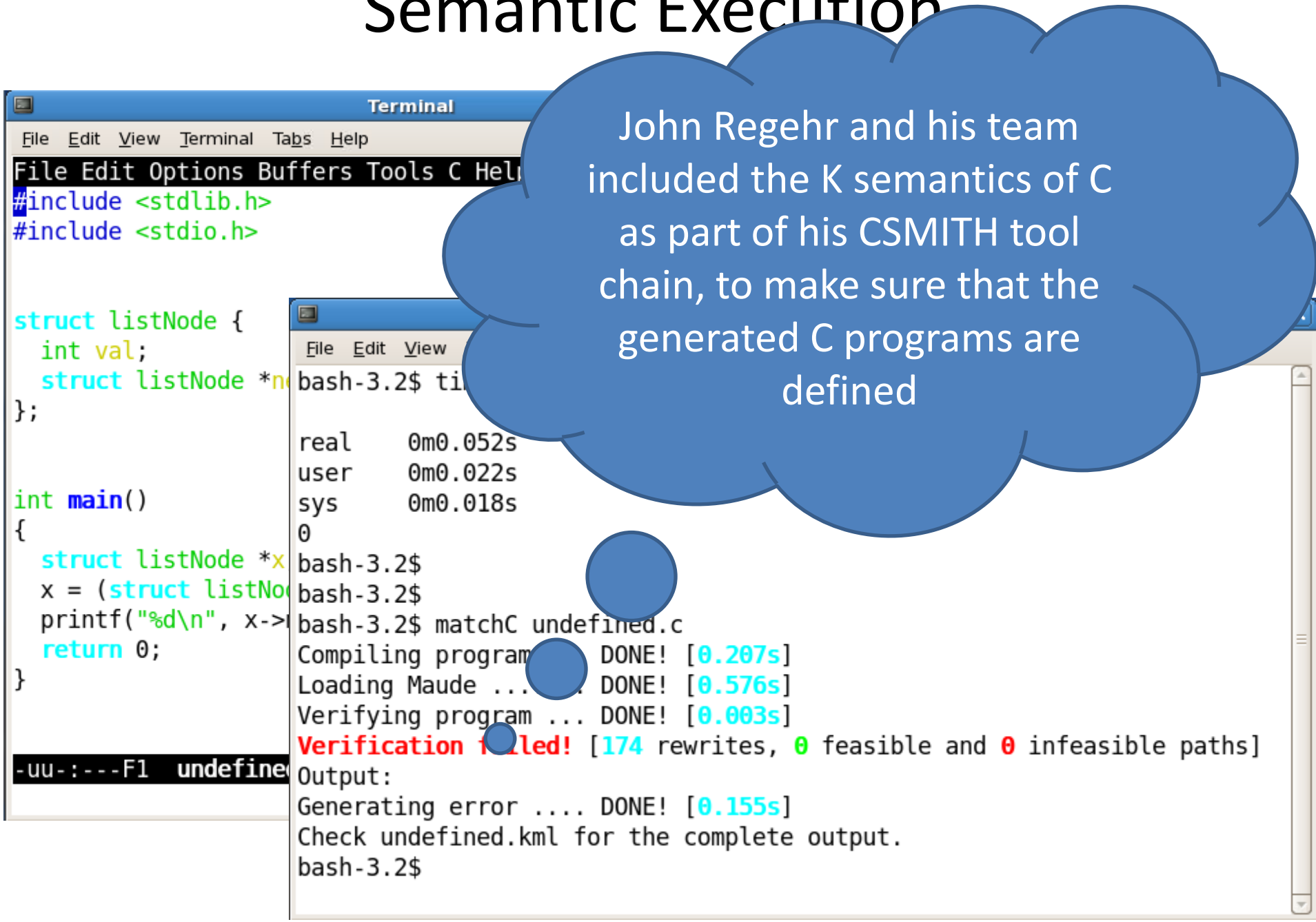

```
<k> $ => return p; </k>
<heap_> list(x,A) => list(p,rev(A)) <_/heap>
```

Partial Correctness

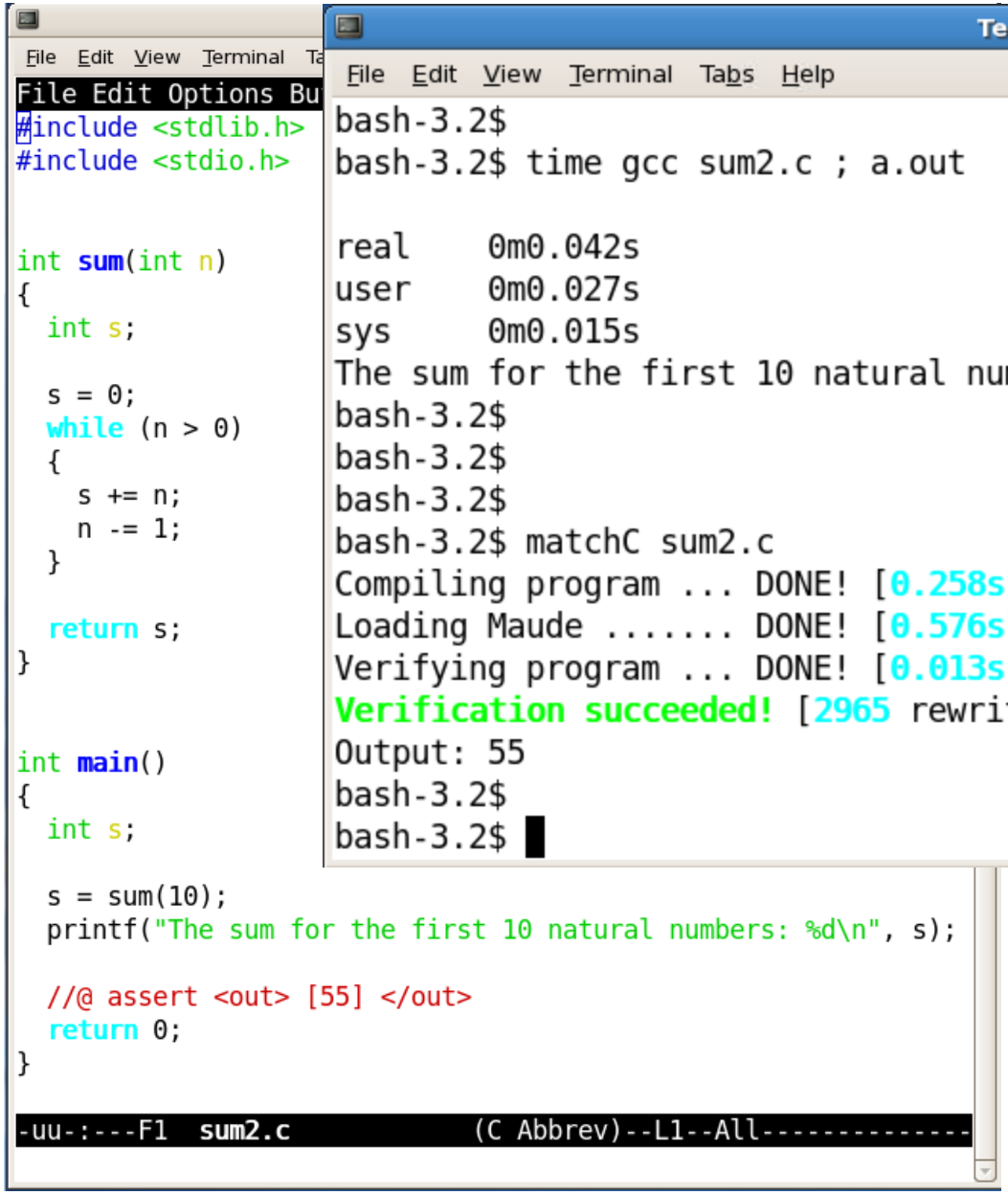
- We have two rewrite relations on configurations
  - $\rightarrow$ given by the language K semantics; safe
  - $\rightarrow$ given by specifications; unsafe, has to be proved
- Idea (simplified for deterministic languages):
  - Pick left $\rightarrow$ right. Show that always left $\rightarrow (\rightarrow \cup \rightarrow)^*$ right modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
  - If left $\rightarrow$ right and "config *matches* left" such that config has a normal form for $\rightarrow$, then "nf(config) *matches* right"

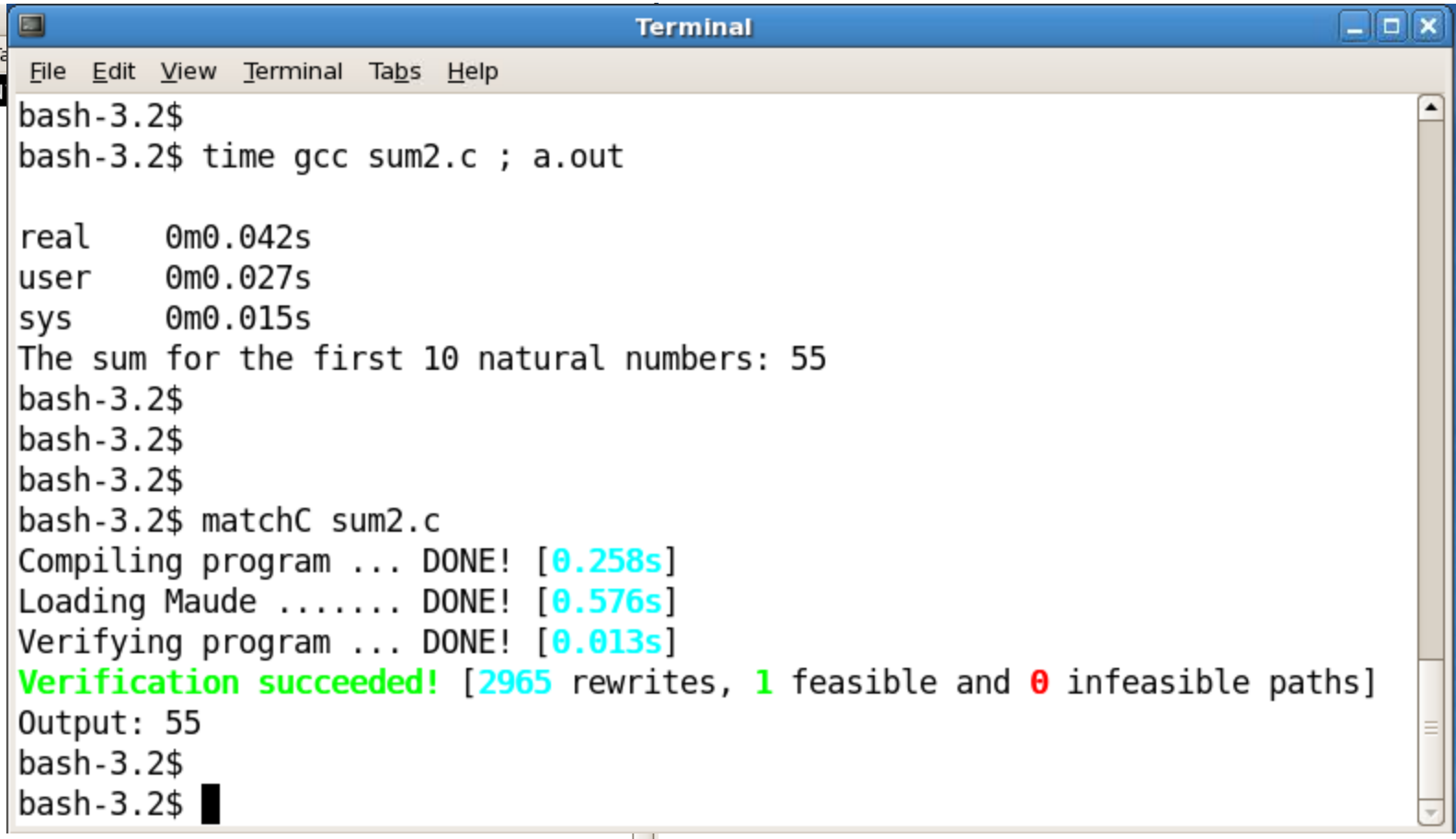
MatchC Tool DEMO
(through slides)

Semantic Execution

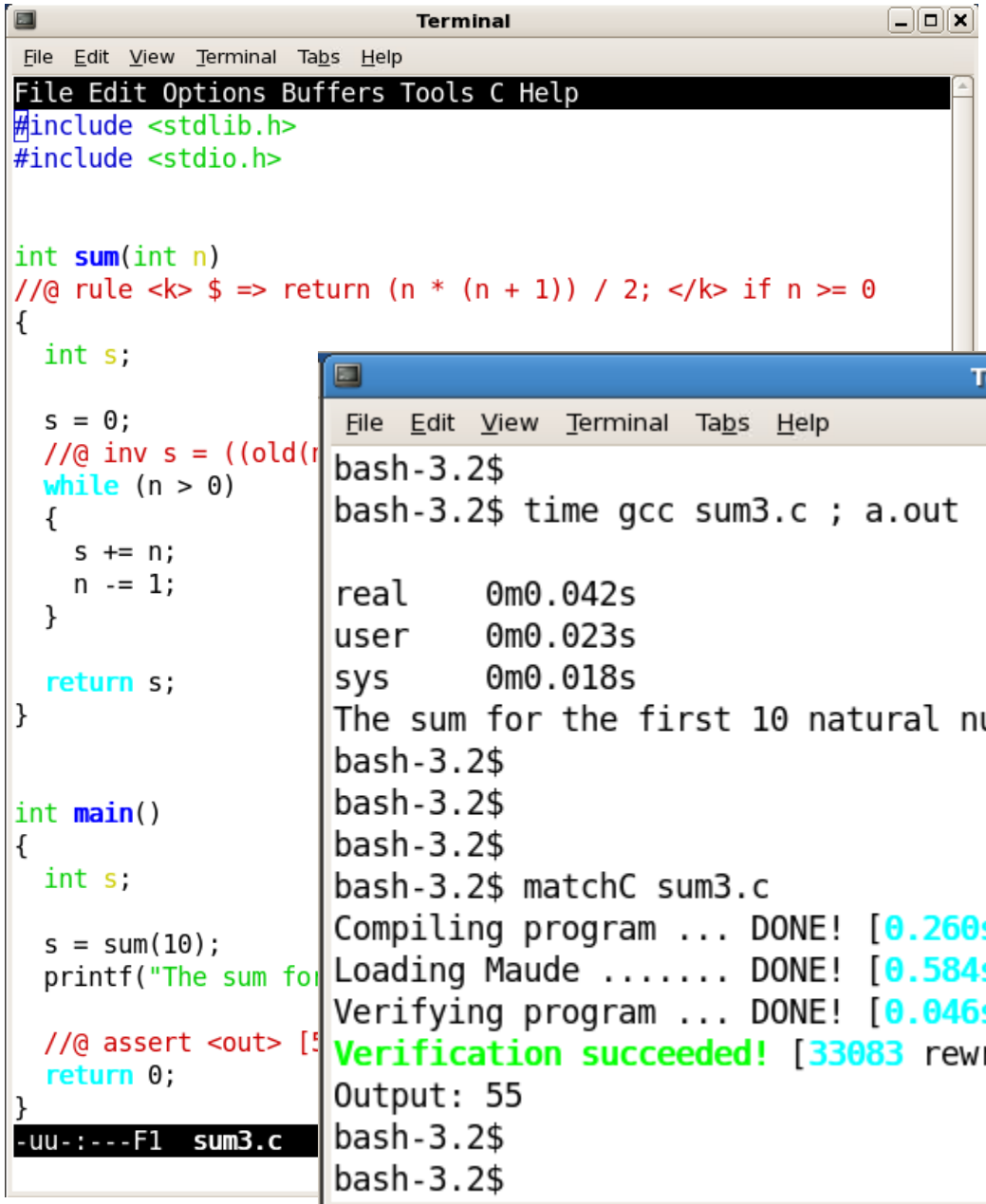


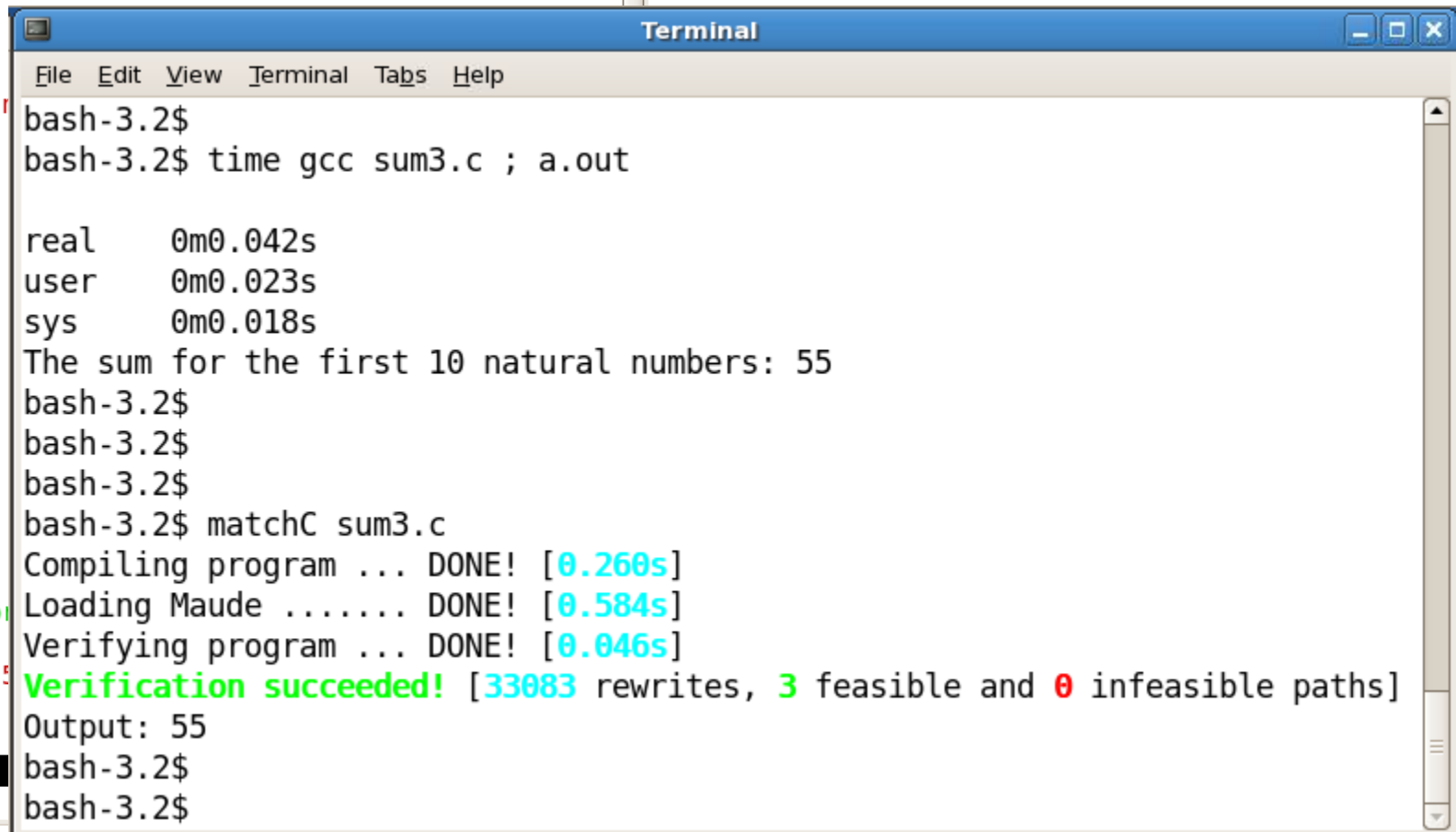
Assertion Checking



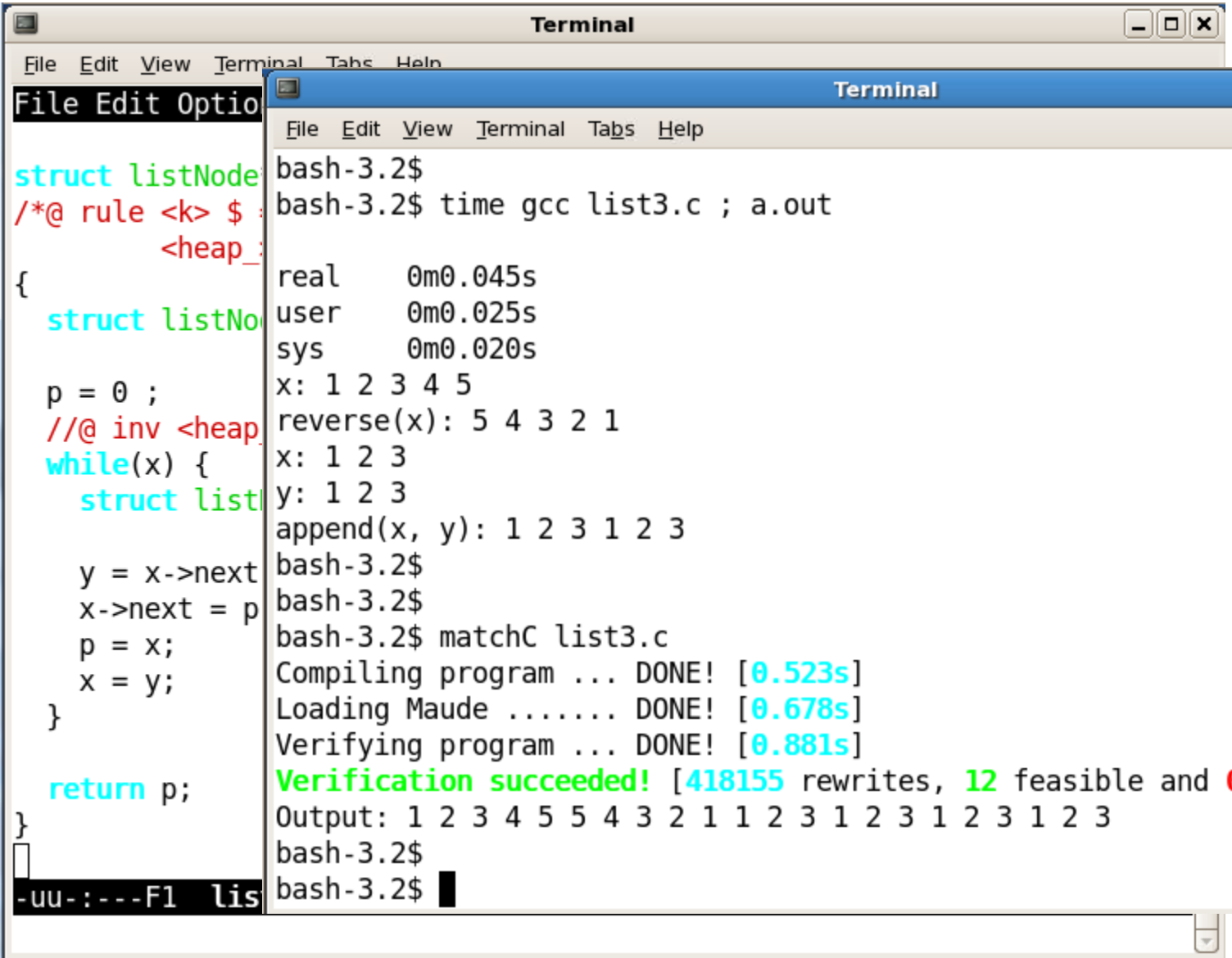


Full Verification

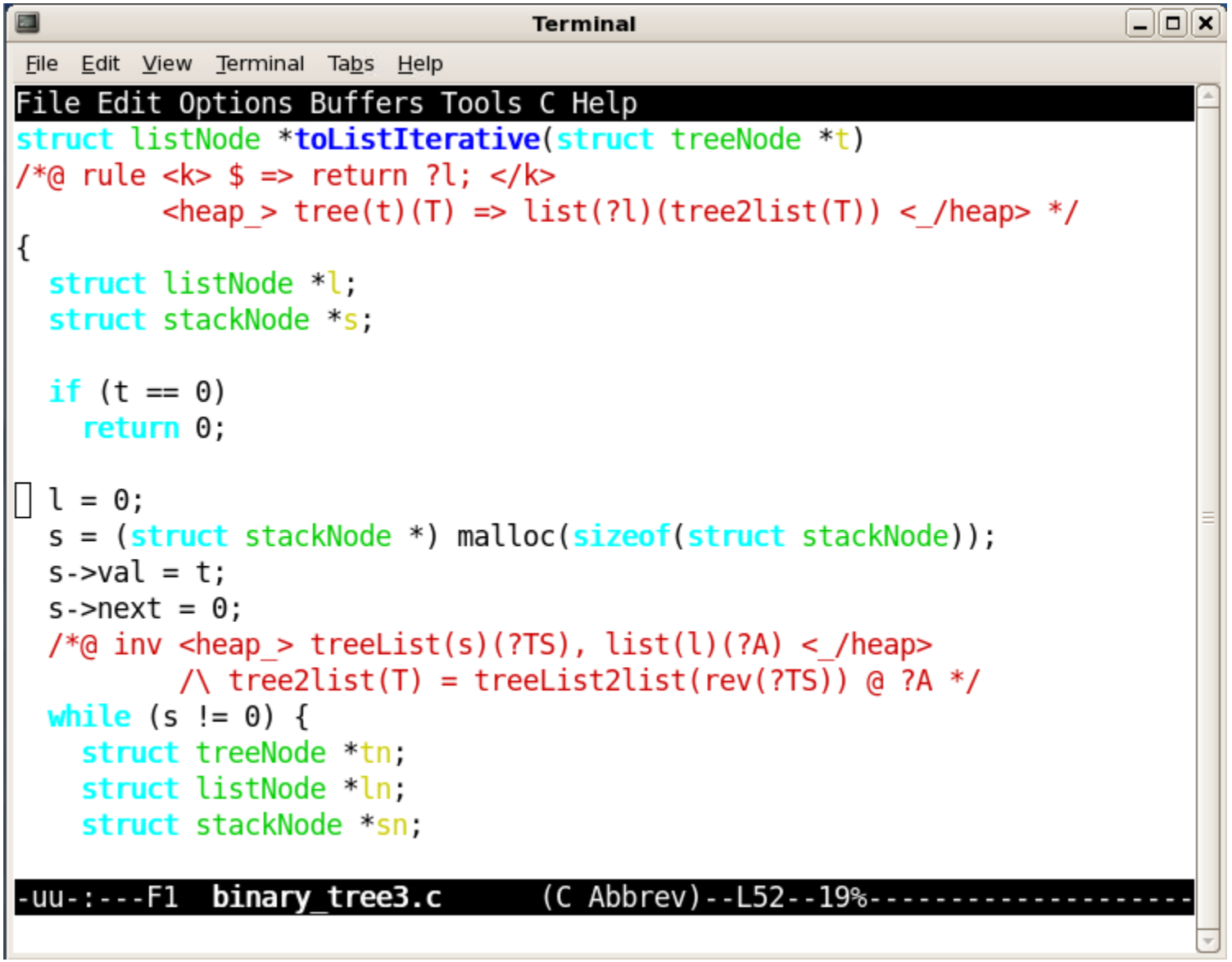




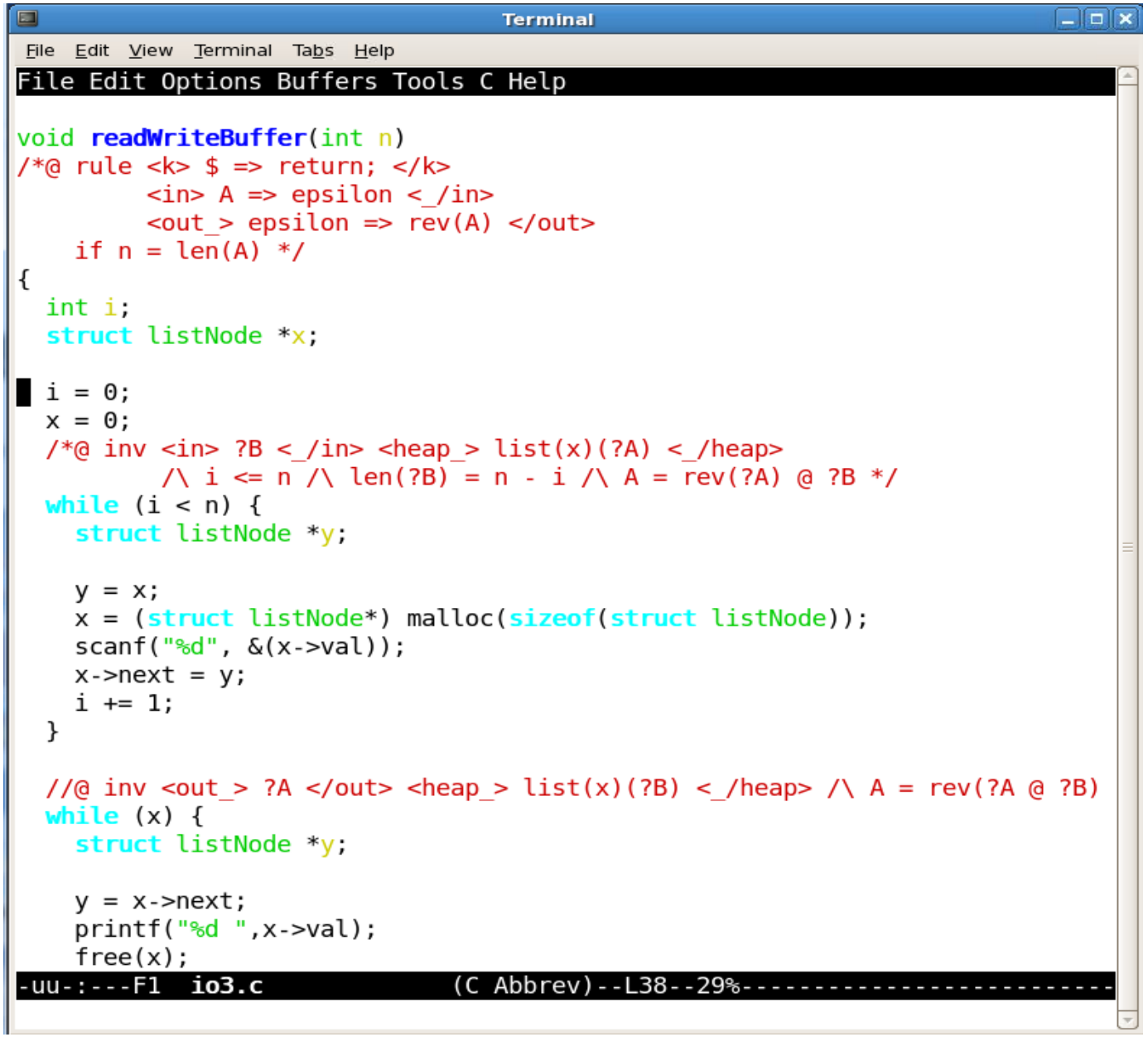
List Examples – Borrowed from SL tools



Beyond Separation Logic Tools



Beyond Separation Logic – I/O



Beyond Separation Logic – Stack Inspection

Terminal

File Edit View Terminal Tabs Help

```
File Edit Options Buffers Tools C Help

void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if n >= 10 \/ in(hd(ids(S)), {main, trusted}) */
{
  printf("%d ", n);
  untrusted(n);
  any(n);
  if (n)
    trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
    if in(trusted, ids(S)) */
{
  printf("%d ", -n);
  if (n)
    any(n - 1);
}

void any(int n)
{
  // untrusted(n);
  if(n > 10)
    // possible security violated if n < 10
    trusted(n - 1);
}
-uu-:---F1  stack_inspection.c      (C Abbrev)--L14--12%-----------------------
```

Conclusions

- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
  - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools